Chomsky's Classification of Grammars

On Certain Formal Properties of Grammars

(N. Chomsky, INFORMATION AND CONTROL 9., 137-167 (1959)

type 0: arbitrary string-rewrite rules

equivalent to Turing machines!

e X b => e X e X => Y

type 1: context sensitive, RHS always larger

O(n)-space Turing machines

a X b => a c X b

type 2: context free - one LHS nonterminal

type 3: regular grammars (regular languages)

Parsing Context-Free Grammars

Decidable even for type 1 grammars,
(by eliminating epsilons - Chomsky 1959)

We choose $O(n^3)$ CYK algorithm - simple

Better complexity possible:

General Context-Free Recognition in Less than Cubic Time, JOURNAL OF COMPUTER AND SYSTEM SCIENCES 10, 308--315 (1975)

- problem reduced to matrix multiplication - n^k for k between 2 and 3

More practical algorithms known:

J. Earley **An efficient context-free parsing algorithm,** Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA (1968)

can be adapted so that it automatically works in quadratic or linear time for better-behaved grammars

CYK Parsing Algorithm

C:
John **Cocke** and Jacob T. Schwartz (1970). Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

Y:
Daniel H. **Younger** (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control* 10(2): 189–208.

K:
T. **Kasami** (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA.

CYK Algorithm Can Handle Ambiguity

Why Parse General Grammars

- General grammars can be ambiguous: for some strings, there are multiple parser trees
- Can be impossible to make grammar unambiguous
- Some languages are more complex than simple programming languages
  - mathematical formulas:

    x = y ∧ z ? (x=y) ∧ z x = (y ∧ z)
  - natural language:

    I saw the man with the telescope.
  - future programming languages

Ambiguity 1

1)

2)

I saw the man with the telescope.

Ambiguity 2

Time flies like an arrow.

Indeed, time passes by quickly.

Those special “time flies” have an “arrow” as their favorite food.

You should regularly measure how fast the flies are flying, using a process that is much like an arrow.

…

Two Steps in the Algorithm

1) Transform grammar to normal form
   called Chomsky Normal Form

2) Parse input using transformed grammar
   dynamic programming algorithm

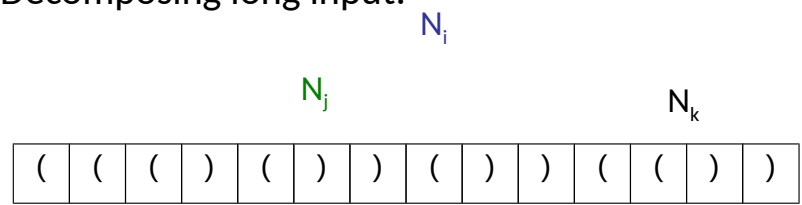
“a method for solving complex problems by breaking them down into simpler steps.
It is applicable to problems exhibiting the properties of overlapping subproblems”

Dynamic Programming to Parse Input

Assume Chomsky Normal Form, 3 types of rules:

$S' \rightarrow \varepsilon \mid S$	(only for the start non-terminal)
$N_i \rightarrow t$	(names for terminals)
$N_i \rightarrow N_j\ N_k$	(just **2** non-terminals on RHS)

Decomposing long input:



find all ways to parse substrings of length 1,2,3,…

Balanced Parentheses Grammar

Original grammar G

$B \rightarrow \varepsilon \mid B\ B \mid (\ B\ )$

Modified grammar in Chomsky Normal Form:

$B1 \rightarrow \varepsilon \mid B\ B \mid O\ M \mid O\ C$

$B \rightarrow B\ B \mid O\ M \mid O\ C$

$M \rightarrow B\ C$

$O \rightarrow$ '('

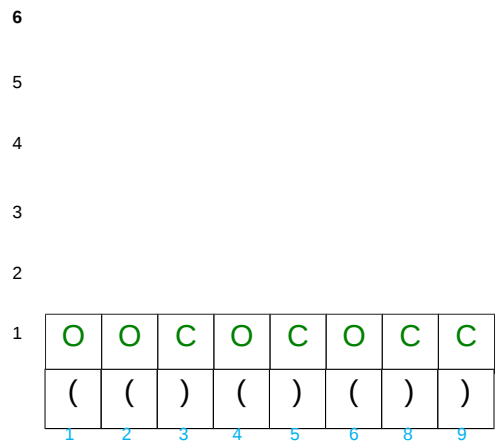
$C \rightarrow$ ')'

Terminals: ( )

Nonterminals: B, B1, O, C, M, B

Parsing an Input

B1 → ε | B B | O M | O C
B → B B | O M | O C
M → B C
O → '('
C → ')'



Algorithm Idea

w_{pq} – substring from p to q

d_{pq} – all non-terminals that could expand to w_{pq}

Initially d_{pp} has $N_{w(p,p)}$

key step of the algorithm:

if $X \rightarrow Y\ Z$ is a rule,
Y is in $d_{p\,r}$, and
Z is in $d_{(r+1)q}$

then put X into d_{pq}

($p <= r < q$),

in increasing value of (q-p)

Algorithm

```
INPUT:  grammar G in Chomsky normal form
            word w to parse using G
OUTPUT: true iff (w in L(G))
N = |w|
var d : Array[N][N]
for p = 1 to N {
   d(p)(p) = {X | G contains X->w(p)}
   for q in {p + 1 .. N} d(p)(q) = {} }
for k = 2 to N // substring length
  for p = 0 to N-k // initial position
    for j = 1 to k-1 // length of first half
      val r = p+j-1; val q = p+k-1;
      for (X::=Y Z) in G
        if Y in d(p)(r) and Z in d(r+1)(q)
           d(p)(q) = d(p)(q) union {X}
return  S in d(0)(N-1)
```

What is the running time as a function of grammar size and the size of input?

O( )

(	(	)	(	)	(	)	)

Number of Parse Trees

Let w denote word ()()()

- it has two parse trees

Give a lower bound on number of parse trees of the word w^n (n is positive integer)

w^5 is the word

()()() ()()() ()()() ()()() ()()()

CYK represents all parse trees compactly

- can re-run algorithm to extract first parse tree, or enumerate parse trees one by one

Conversion to Chomsky Normal Form (CNF)

Steps: (not in the optimal order)

- remove unproductive symbols
- remove unreachable symbols
- remove epsilons (no non-start nullable symbols)
- remove single non-terminal productions
  (unit productions) X::=Y
- reduce arity of every production to less than two
- make terminals occur alone on right-hand side

1) Unproductive non-terminals

What is funny about this grammar:

```
stmt ::=  identifier := identifier
        | while (expr) stmt
        | if (expr) stmt else stmt
expr ::= term + term | term – term
term ::= factor * factor
factor ::= ( expr )
```

There is no derivation of a sequence of tokens from expr

In every step will have at least one expr, term, or factor

If it cannot derive sequence of tokens we call it *unproductive*

1) Unproductive non-terminals

Productive symbols are obtained using these two rules (what remains is unproductive)

- Terminals are productive
- If $X ::= s_1\ s_2 \ldots s_n$ is a rule and each s_i is productive then X is productive

Delete unproductive symbols.

The language recognized by the grammar will not change

2) Unreachable non-terminals

What is funny about this grammar with start symbol ‘program’

program ::= stmt | stmt program
stmt ::= assignment | whileStmt
assignment ::= expr = expr
ifStmt ::= if (expr) stmt else stmt
whileStmt ::= while (expr) stmt
expr ::= identifier

No way to reach symbol ‘ifStmt’ from ‘program’

Can we formulate rules for reachable symbols ?

2) Unreachable non-terminals

Reachable terminals are obtained using the following rules (the rest are unreachable)

- starting non-terminal is reachable (program)
- If $X ::= s_1\ s_2 \ldots s_n$ is rule and X is reachable then every non-terminal in $s_1\ s_2 \ldots s_n$ is reachable

Delete unreachable nonterminals and their productions

3) Removing Empty Strings

Ensure only top-level symbol can be nullable

```
program ::= stmtSeq
stmtSeq ::= stmt | stmt ; stmtSeq
stmt ::= “” | assignment | whileStmt | blockStmt
blockStmt ::= { stmtSeq }
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
expr ::= identifier
```

How to do it in this example?

3) Removing Empty Strings - Result

program ::= “” | stmtSeq
stmtSeq ::= stmt| stmt ; stmtSeq |
| ; stmtSeq | stmt ; | ;
stmt ::= assignment | whileStmt | blockStmt
blockStmt ::= { stmtSeq } | { }
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
whileStmt ::= while (expr)
expr ::= identifier

3) Removing Empty Strings - Algorithm

$O(2^n)$

3) Removing Empty Strings

- Since stmtSeq is nullable, the rule
  blockStmt ::= { stmtSeq }
  gives
  blockStmt ::= { stmtSeq } | { }
- Since stmtSeq and stmt are nullable, the rule
  stmtSeq ::= stmt | stmt ; stmtSeq
  gives
  stmtSeq ::= stmt | stmt ; stmtSeq
  | ; stmtSeq | stmt ; | ;

4) Eliminating unit productions

- Single production is of the form

  X ::=Y

where X,Y are non-terminals

```
program ::= stmtSeq
stmtSeq ::= stmt
          | stmt ; stmtSeq
stmt ::= assignment | whileStmt
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
```

4) Unit Production Elimination Algorithm

- If there is a unit production

  X ::=Y put an edge (X,Y) into graph

- If there is a path from X to Z in the graph, and there is rule Z ::= s_1 s_2 … s_n then add rule

  X ::= s_1 s_2 … s_n

At the end, remove all unit productions.

4) Eliminate unit productions - Result

```
program ::= expr = expr | while (expr) stmt
          | stmt ; stmtSeq
stmtSeq ::= expr = expr | while (expr) stmt
          | stmt ; stmtSeq
stmt ::= expr = expr | while (expr) stmt
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
```

5) Reducing Arity:
No more than 2 symbols on RHS

stmt ::= while (expr) stmt

becomes

stmt ::= while $stmt_1$

$stmt_1$ ::= ($stmt_2$

$stmt_2$ ::= expr $stmt_3$

$stmt_3$ ::=) stmt

6) A non-terminal for each terminal

stmt ::= while (expr) stmt

becomes

stmt ::= N_{while} $stmt_1$

$stmt_1$ ::= $N_{(}$ $stmt_2$

$stmt_2$ ::= expr $stmt_3$

$stmt_3$ ::= $N_{)}$ stmt

N_{while} ::= while

$N_{(}$ ::= (

$N_{)}$ ::=)

Order of steps in conversion to CNF

1. remove unproductive symbols (optional)
2. remove unreachable symbols (optional)
3. make terminals occur alone on right-hand side
4. Reduce arity of every production to <= 2
5. remove epsilons
6. remove unit productions X::=Y
7. unproductive symbols
8. unreachable symbols

- What if we swap the steps 4 and 5 ?
  - Potentially exponential blow-up in the # of productions

Ordering of
Unreachable / Unproductive symbols

First Unreachable then Unproductive

S := B C | “”
C := D
D := a
R := r

S := B C | “”
C := D
D := a

S := “”
C := D
D := a

First Unproductive then Unreachable

S := B C | “”
C := D
D := C
R := r

S := “”
C := D
D := a
R := r

S := “”

Alternative to Chomsky form

We need not go all the way to Chomsky form
it is possible to directly parse arbitrary grammar
Key steps: (not in the optimal order)
- reduce arity of every production to less than two
(otherwise, worse than cubic in string input size)

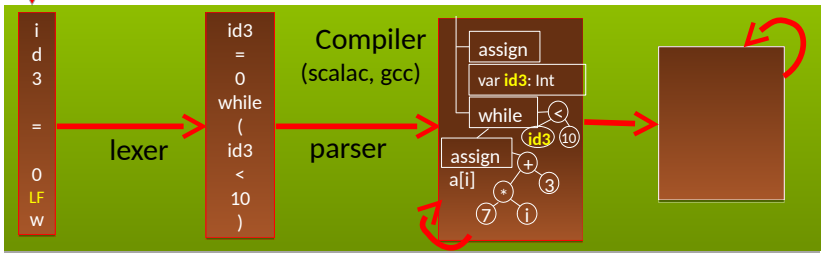
Can be less efficient in grammar size, but still works

More algorithms for arbitrary grammars are variations:
Earley's parsing algorithm (Earley, CACM 1970)
GLR parsing algorithm (Lang, ICALP 1974, Deterministic Techniques for Efficient Non-Deterministic Parsers)
GLL algorithm

```
Id3 = 0
while (id3 < 10) {
  println("",id3);
  id3 = id3 + 1 }
```

source code: sequence of characters

after each analysis the compiler has a better "understanding" of the input program; can report more subtle errors



characters | words (tokens) | trees

Name Analysis:

making sense of trees;
converting them into **graphs**:
connect identifier **uses** and **declarations**

Reporting Errors

Errors Detected So Far

- File input: file does not exist
- Lexer: unknown token, string not closed before end of file, ...
- Parser: syntax error - unexpected token, cannot parse given non-terminal
- Name analyzer: unknown identifier
- Type analyzer:
  applying function to arguments of wrong type
- Data-flow analyzer:
  variable read before written, division by zero

Name Analysis Problems Reported: 1

- a class is defined more than once:

  class A { ...} class B { ... } class A { ... }
- a variable is defined more than once:

  int x; int y; int x;
- a class member is overriden without **override** keyword:

  class A { int x; ... } class B extends A { int x; ... }
- a method is **overloaded** (forbidden in Tool):

  class A { def f(B x) {} def f(C x) {} ... }
- a method argument is shadowed by a local variable declaration (forbidden in Java, Tool):

  def (x:Int) { var x : Int; ...}
- two method arguments have the same name:

  def (x:Int,y:Int,x:Int) { ... }

Name Analysis Problems Reported: 2

- a class name is used as a symbol (as parent class or type, for instance) but is not declared:

  class A extends Objekt {}

- an identifier is used as a variable but is not declared:

  def(amount:Int) { total = total + ammount }

- the inheritance graph has a cycle:

  class A extends B {}
  class B extends C {}
  class C extends A

To make it efficient and clean to check for such errors, we associate mapping from each identifier to the symbol that the identifier represents.

- We use Map data structures to maintain this mapping
- The rules that specify how declarations are used to construct such maps are given by ***scoping* rules of the programming language.**

Storing and Using Tree Positions

Showing Good Errors with Syntax Trees

Suppose we have undeclared variable ‘i’ in a program of 100K lines

Which error message would you prefer to see from the compiler?

- An ocurrence of variable ‘i’ not declared (which variable? where?)
- An ocurrence of variable ‘i’ in procedure P not declared
- Variable ‘i’ undeclared at line 514, position 12 (and IDE points you there)

How to emit this error message if we only have a syntax trees?

- Abstract syntax tree nodes store positions within file
- For identifier nodes: allows reporting variable uses
  - Variable 'i' in line 11, column 5 undeclared
- For other nodes, supports useful for type errors, e.g. could report for `(x + y) * (!ok)`
  - Type error in line 13,
  - expression in line 13, column 11-15, has type **Bool**, expected **Int** instead

Showing Good Errors with Syntax Trees

Constructing trees with positions:

- Lexer records positions for tokens
- Each subtree in AST corresponds to some parse tree, so it has first and last token
- Get positions from those tokens
- Save these positions in the constructed tree

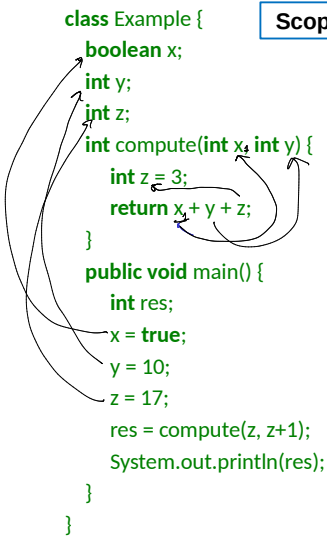
What is important is to save information for leaves

- information for other nodes can often be approximated using information in the leaves

Continuing Name Analysis:

Scope of Identifiers

Example: find program result, symbols, scopes



Scope of a variable = part of the program where it is visible

Draw an arrow from occurrence of each identifier to the point of its declaration.

For each declaration of identifier, identify where the identifier can be referred to (its scope).

Name analysis:

- computes those arrows
  - = maps, partial functions (math)
  - = environments (PL theory)
  - = symbol table (implementation)
- report some simple semantic errors

We usually introduce **symbols** for things denoted by identifiers.
Symbol tables map identifiers to symbols.

Usual **static** scoping: What is the result?

```
class World {
  int sum;
  int value;
  void add() {
     sum = sum + value;
     value = 0;
  }
  void main() {
      sum = 0;
      value = 10;
      add();
      if (sum % 3 == 1) {
        int value;
        value = 1;
        add();
        print("inner value = ", value);  1
        print("sum = ", sum);  10
      }
      print("outer value = ", value);  0
   }
}
```

Identifier refers to the symbol that was declared “closest” to the place **in program structure** (thus "static").

We will assume static scoping unless otherwise specified.

Renaming Statically Scoped Program

```
class World {
  int sum;
  int value;
  void add(int foo) {
     sum = sum + value;
     value = 0;
  }
  void main() {
      sum = 0;
      value = 10;
      add();
      if (sum % 3 == 1) {
        int value1;
        value1 = 1;
        add(); // cannot change value1
        print("inner value = ", value1); 1
        print("sum = ", sum);   10
      }
      print("outer value = ", value);   0
   }
}
```

Identifier refers to the symbol that was declared “closest” to the place **in program structure** (thus "static").

We will assume static scoping unless otherwise specified.

Property of static scoping:
Given the entire program, we can **rename variables** to avoid any shadowing (**make** all vars **unique**!)

Dynamic scoping: What is the result?

```
class World {
  int sum;
  int value;
  void add() {
     sum = sum + value;
     value = 0;
  }
  void main() {
      sum = 0;
      value = 10;
      add();
      if (sum % 3 == 1) {
        int value;
        value = 1;
        add();
        print("inner value = ", value);  0
        print("sum = ", sum);   11
      }
      print("outer value = ", value);   0
   }
}
```

Symbol refers to the variable that was most **recently declared within program execution.**

Views variable declarations as executable statements that establish which symbol is considered to be the ‘current one’. (Used in old LISP interpreters.)

Translation to normal code: access through a dynamic environment.

Dynamic scoping translated using global map, working like stack

```
class World {
  int sum;
  int value;
  void add() {
     sum = sum + value;
     value = 0;
  }
  void main() {
      sum = 0;
      value = 10;
      add();
      if (sum % 3 == 1) {
        int value;
        value = 1;
        add();
        print("inner value = ", value);   0
        print("sum = ", sum);   11
      }
      print("outer value = ", value);   0
   }
}
```

```
class World {
  pushNewDeclaration('sum);
  pushNewDeclaration('value);
  void add(int foo) {
     update('sum, lookup('sum) + lookup('value));
     update('value, 0);
  }
  void main() {
      update('sum, 0);
      update('value,10);
      add();
      if (lookup('sum) % 3 == 1) {
        pushNewDeclaration('value);
        update('value, 1);
        add();
        print("inner value = ", lookup('value));
        print("sum = ", lookup('sum));
        popDeclaration('value)
      }
      print("outer value = ", lookup('value));
   }
}
```

Object-oriented programming has scope for each object, so we have a nice controlled alternative to dynamic scoping (objects give names to scopes).

Good Practice for Scoping

- Static scoping is almost universally accepted in modern programming language design
- It is the approach that is usually easier to reason about and easier to **compile**, since we do not have names at compile time and compile each code piece separately
- Still, various ad-hoc language designs emerge and become successful
  - LISP implementations took dynamic scoping since it was simpler to implement for higher-order functions
  - Javascript

How the **symbol map** changes in case of **static** scoping

```
class World {
  int sum;  int value;
  // value → int, sum → int
  void add(int foo) {
      // foo → int, value → int, sum → int
      string z;
      // z → string, foo → int, value → int, sum → int
      sum = sum + value; value = 0;
  }
  // value → int, sum → int
  void main(string bar) {
      // bar → string, value → int, sum → int
      int y;
      // y → int, bar → string, value → int, sum → int
      sum = 0;
      value = 10;
      add();
      // y → int, bar → string, value → int, sum → int
      if (sum % 3 == 1) {
        string value;
        // value → string, y → int, bar → string, sum → int
        value = 1;
        add();
        print("inner value = ", value);
        print("sum = ", sum); }
      // y → int, bar → string, value → int, sum → int
      print("outer value = ", value);
} }
```

Outer declaration
int value is shadowed by
inner declaration **string value**

Map becomes bigger as
we enter more scopes,
later becomes smaller again
Imperatively: need to make
maps bigger, later smaller again.
Functionally: immutable maps,
keep old versions.

Representing Data

- In Java, the standard model is a mutable graph of objects
- It seems natural to represent references to symbols using mutable fields (initially null, **resolved** during name analysis)
- Alternative way is functional
  - store the **backbone** of the graph as a algebraic data type (immutable)
  - pass around a map linking from identifiers to their declarations
- Note that a field **class** A { **var** f:T } is like f: Map[A,T]

Symbol Table (Γ) Contents

- Map identifiers to the symbol with relevant information about the identifier
- All information is derived from syntax tree - symbol table is for efficiency
  - in old one-pass compilers there was only symbol table, no syntax tree
  - in modern compiler: we could always go through entire tree, but symbol table can give faster and easier access to the part of syntax tree, or some additional information
- Goal: efficiently supporting phases of compiler
- In the name analysis phase:
  - finding which identifier refers to which definition
  - we store *definitions*

- What kinds of things can we define? What do we need to know for each ID?

  variables (globals, fields, parameters, locals):
  - need to know types, positions - for error messages
  - later: memory layout. To compile x.f = y into memcopy(addr_y, addr_x+6, 4)
    - e.g. 3rd field in an object should be stored at offset e.g. +6 from the address of the object
    - the size of data stored in x.f is 4 bytes
  - sometimes more information explicit: whether variable local or global

    methods, functions, classes: recursively have with their own symbol tables

Functional: Different Points, Different Γ

```
class World {
  int sum;
  void add(int foo) {
    sum = sum + foo;
  }
  void sub(int bar) {
    sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0\,[foo := int]$

Γ_0

$\Gamma_1 = \Gamma_0\,[bar := int]$

Imperative Way: Push and Pop

```
class World {
  int sum;
  void add(int foo) {
    sum = sum + foo;
  }
  void sub(int bar) {
    sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0\ [foo := int]$ change table, record change

Γ_0 revert changes from table

$\Gamma_1 = \Gamma_0\ [bar := int]$ change table, record change

revert changes from table

Imperative Symbol Table

- Hash table, mutable Map[ID,Symbol]
- Example:
  - hash function into array
  - array has linked list storing (ID,Symbol) pairs
- Undo stack: to enable entering and leaving scope
- Entering new scope (function,block):
  - add beginning-of-scope marker to undo stack
- Adding nested declaration (ID,sym)
  - lookup old value symOld, push old value to undo stack
  - insert (ID,sym) into table
- Leaving the scope
  - go through undo stack until the marker, restore old values

Functional: Keep Old Version

```
class World {
  int sum;
  void add(int foo) {
     sum = sum + foo;
  }
  void sub(int bar) {
     sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0\ [foo := int]$

create new Γ_1, keep old Γ_0

Γ_0

$\Gamma_2 = \Gamma_0\ [bar := int]$

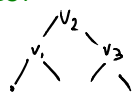
create new Γ_2, keep old Γ_0

Functional Symbol Table Implemented

- Typical: Immutable Balanced Search Trees

```
sealed abstract class BST
case class Empty() extends BST
case class Node(left: BST, value: Int, right: BST) extends BST
```

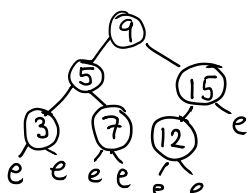
Simplified. In practice, BST[A], store Int key and value A



- Updating returns new map, keeping old one
  - lookup and update both *log(n)*
  - update creates new path (copy *log(n)* nodes, share rest!)
  - memory usage acceptable

Lookup

```
def contains(key: Int, t : BST): Boolean = t match {
   case Empty() => false
   case Node(left,v,right) => {
      if (key == v) true
      else if (key < v) contains(key, left)
      else contains(key, right)
   }
}
```

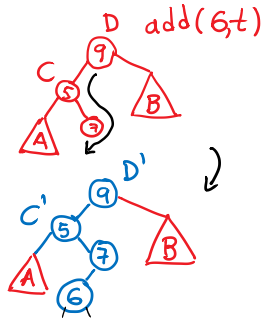


Running time bounded by tree height.

contains(6,t) ?

Insertion

```
def add(x : Int, t : BST) : Node = t match {
  case Empty() => Node(Empty(),x,Empty())
  case t @ Node(left,v,right) => {
    if (x < v) Node(add(x, left), v, right)
    else if (x==v) t
    else Node(left, v, add(x, right))
  }
}
```

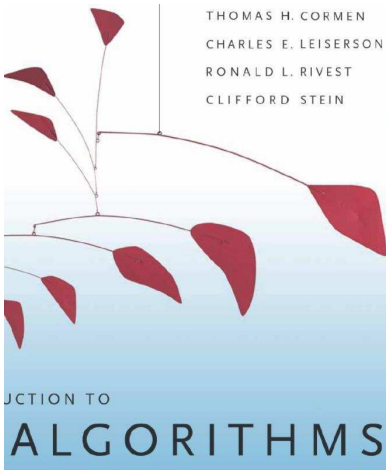


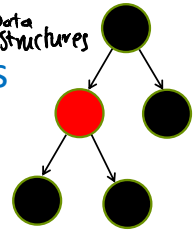
Both add(x,t) and t remain accessible.

Running time and newly allocated nodes bounded by tree height.

Chris Okasaki : Purely Functional Data Structures

Balanced Trees: Red-Black Trees





Balanced Tree: Red Black Tree

Goals:

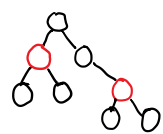
- ensure that tree height remains at most log(size)

add(1,add(2,add(3,...add(n,Empty())...))) ~ linked list

- preserve efficiency of individual operations:
  rebalancing arbitrary tree: could cost O(n) work

Solution: maintain mostly balanced trees: height still O(log size)

```
sealed abstract class Color
case class Red() extends Color
case class Black() extends Color
```



```
sealed abstract class Tree
case class Empty() extends Tree
case class Node(c: Color,left: Tree,value: Int, right: Tree)
                extends Tree
```

Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following ***red-black properties***:

balanced tree constraints

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

From 4. and 5.: tree height is O(log size).

Analysis is similar for mutable and immutable trees.

for immutable trees: see book by Chris Okasaki

Balancing

```
def balance(c: Color, a: Tree, x: Int, b: Tree): Tree = (c,a,x,b) match {
  case (Black(),Node(Red(),Node(Red(),a,xV,b),yV,c),zV,d) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
```


```
  case (Black(),Node(Red(),a,xV,Node(Red(),b,yV,c)),zV,d) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),Node(Red(),b,yV,c),zV,d)) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),b,yV,Node(Red(),c,zV,d))) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (c,a,xV,b) => Node(c,a,xV,b)
}
```

Insertion

```
def add(x: Int, t: Tree): Tree = {
  def ins(t: Tree): Tree = t match {
    case Empty() => Node(Red(),Empty(),x,Empty())
    case Node(c,a,y,b) =>
      if (x < y) balance(c, ins(a), y, b)
      else if (x == y) Node(c,a,y,b)
      else balance(c,a,y,ins(b))
  }
  makeBlack(ins(t))
}
def makeBlack(n: Tree): Tree = n match {
    case Node(Red(),l,v,r) => Node(Black(),l,v,r)
    case _ => n
}
```

Modern object-oriented languages (e.g. Scala) support abstraction and functional data structures. Just use Map from Scala.

Exercise

Determine the output of the following program assuming static and dynamic scoping. Explain the difference, if there is any.

```
object MyClass {
  val x = 5
  def foo(z: Int): Int = { x + z }
  def bar(y: Int): Int = {
     val x = 1; val z = 2
     foo(y)
  }
  def main() {
    val x = 7
    println(foo(bar(3)))
  }
}
```